CSIT5100 Assignment 1 Report

Name: Hou Jiefeng (侯杰锋)

Student number: 20361723

Introduction

This report outlines the unit testing brief for a hotel management system called HMS. The HMS helps a hotel manager to perform customer's check-in/check-out, and room rate/discount management tasks. This report consists of basic introduction of testing system, and analysis of the testing procedures and results.

The Basic Structure of HMS

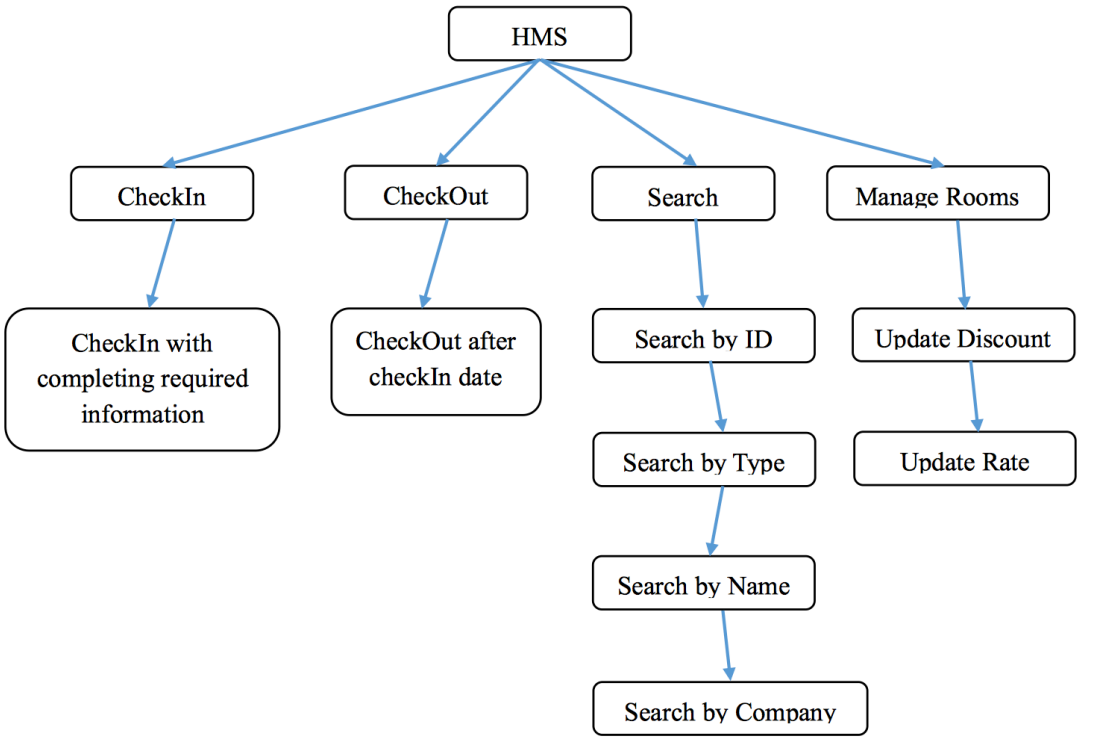


Figure 1: Basic Structure of HMS

The HMS is operated using a graphical user interface (GUI), so hotel manager can complete the tasks with few basic operations instead of complexity commands. In general, the HMS consists of four different user interface panels, including check-in

panel, check-out panel, search panel and manage room panel. Also, hotel manager can change theme and switch different panels in menu of HMS.

In check-in panel, hotel manager can complete a check-in task with inputting all required customer's information, including ID, name, type, company, check-in date, data service required, and Ethernet address. In check-out panel, hotel manager can help customer to check-out by a correct check-out date. In search panel, hotel manager can search for a specific guest in the hotel by ID, name, company and type. In manage room panel, hotel manager can update the rate and discount for a selected room.

Test Procedures

A suite of JUnit test cases should construct under JUnit 4 framework, and should be put into the "test" folder under the default package and added to "*CSIT5100_TestMain*". In addition, using EclEmma to assess the code coverage is necessary.

Test Case Structure

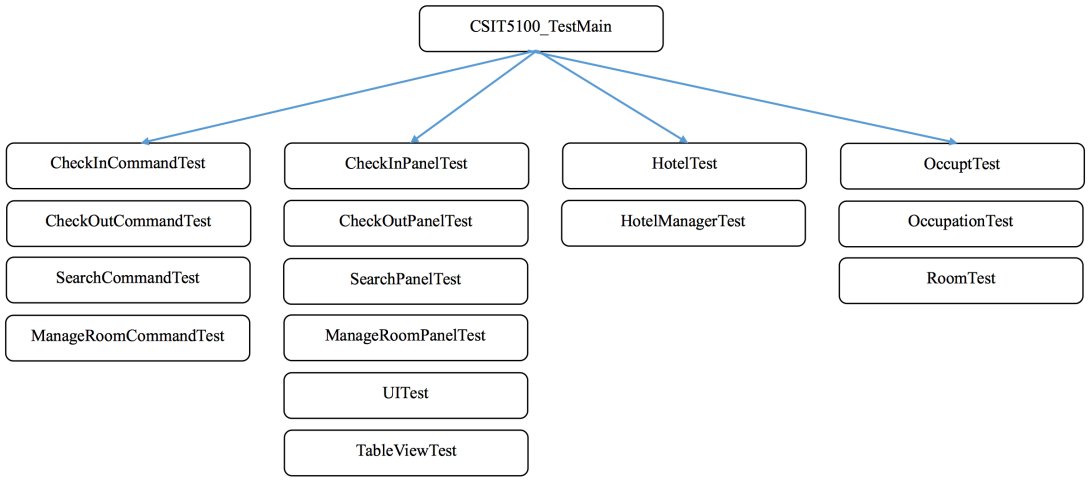


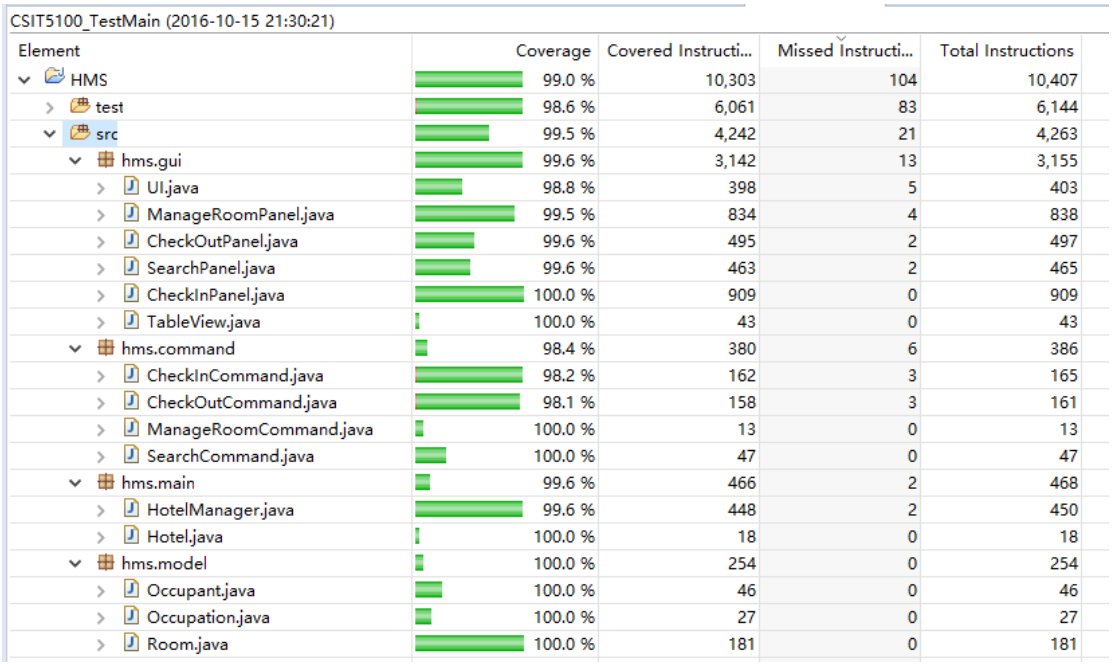
Figure 2: Test Case Structure

The JUnit test cases for HMS are represented by total 15 test classes, and registered in the test suit "*CSIT5100_TestMain*". All in all, there is a one-to-one correspondence between each test class and each Java class of HMS. Furthermore, each method of each class of HMS is tested by an individual test case of the corresponding test class. But the interface "Command" implemented by "*CheckInCommand*", "*CheckOutCommand*",

"*ManageRoomCommand*" and "*SearchCommand*", is unnecessary to construct a test case since there is no method to be tested in this interface.

Statement Coverage and Branch Coverage (on Windows OS)

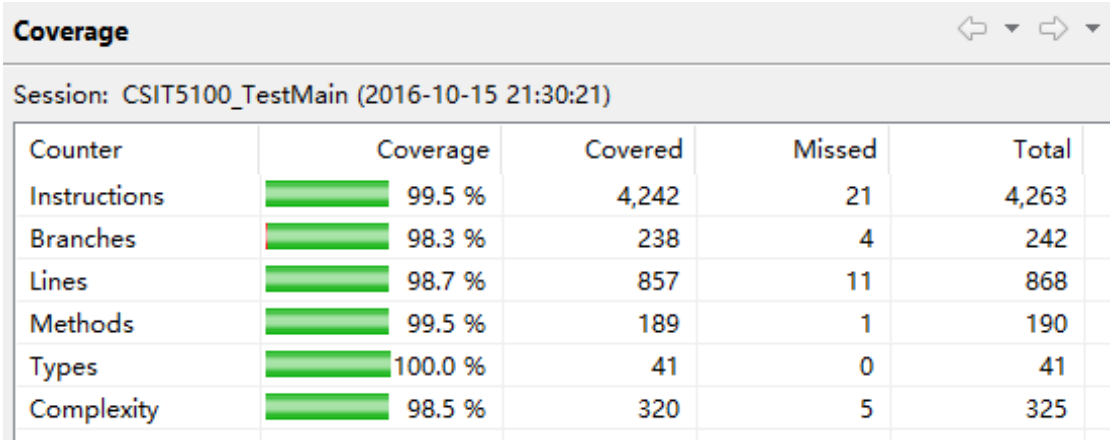
The test cases can achieve 99.5% statement coverage, and 98.3% branch coverage for the source codes of HMS.



CSIT5100_TestMain (2016-10-15 21:30:21)

Element	Coverage	Covered Instructi...	Missed Instructi...	Total Instructions
HMS	99.0 %	10,303	104	10,407
test	98.6 %	6,061	83	6,144
src	99.5 %	4,242	21	4,263
hms.gui	99.6 %	3,142	13	3,155
UI.java	98.8 %	398	5	403
ManageRoomPanel.java	99.5 %	834	4	838
CheckOutPanel.java	99.6 %	495	2	497
SearchPanel.java	99.6 %	463	2	465
CheckInPanel.java	100.0 %	909	0	909
TableView.java	100.0 %	43	0	43
hms.command	98.4 %	380	6	386
CheckInCommand.java	98.2 %	162	3	165
CheckOutCommand.java	98.1 %	158	3	161
ManageRoomCommand.java	100.0 %	13	0	13
SearchCommand.java	100.0 %	47	0	47
hms.main	99.6 %	466	2	468
HotelManager.java	99.6 %	448	2	450
Hotel.java	100.0 %	18	0	18
hms.model	100.0 %	254	0	254
Occupant.java	100.0 %	46	0	46
Occupation.java	100.0 %	27	0	27
Room.java	100.0 %	181	0	181

Figure 3: Statement Coverage (on Windows OS)



Coverage

Session: CSIT5100_TestMain (2016-10-15 21:30:21)

Counter	Coverage	Covered	Missed	Total
Instructions	99.5 %	4,242	21	4,263
Branches	98.3 %	238	4	242
Lines	98.7 %	857	11	868
Methods	99.5 %	189	1	190
Types	100.0 %	41	0	41
Complexity	98.5 %	320	5	325

Figure 4: Branch Coverage (on Windows OS)

Statement Coverage and Branch Coverage (on Mac OS)

The test cases can achieve 98.5% statement coverage, and 94.6% branch coverage for the source codes of HMS. In addition, this report (infeasible statement part) will discuss the reason why Windows OS can achieve higher statement and branch coverage than Mac OS.

CSIT5100_TestMain (2016-10-14 20:04:50)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
HMS	98.6 %	10,240	150	10,390
src	98.5 %	4,200	63	4,263
hms.gui	98.3 %	3,100	55	3,155
SearchPanel.java	96.6 %	449	16	465
CheckInPanel.java	96.9 %	881	28	909
UI.java	98.8 %	398	5	403
ManageRoomPanel.java	99.5 %	834	4	838
CheckOutPanel.java	99.6 %	495	2	497
TableView.java	100.0 %	43	0	43
hms.command	98.4 %	380	6	386
CheckOutCommand.java	98.1 %	158	3	161
CheckInCommand.java	98.2 %	162	3	165
ManageRoomCommand.java	100.0 %	13	0	13
SearchCommand.java	100.0 %	47	0	47
hms.main	99.6 %	466	2	468
HotelManager.java	99.6 %	448	2	450
Hotel.java	100.0 %	18	0	18
hms.model	100.0 %	254	0	254
Occupant.java	100.0 %	46	0	46
Occupation.java	100.0 %	27	0	27
Room.java	100.0 %	181	0	181
test	98.6 %	6,040	87	6,127

Figure 5: Statement Coverage (on Mac OS)

Coverage

Session: CSIT5100_TestMain (2016-10-14 20:04:50)

Counter	Coverage	Covered	Missed	Total
Instructions	98.5 %	4,200	63	4,263
Branches	94.6 %	229	13	242
Lines	97.7 %	848	20	868
Methods	99.5 %	189	1	190
Types	100.0 %	41	0	41
Complexity	96.6 %	314	11	325

Figure 6: Branch Coverage (on Mac OS)

High Code Coverage

It is no doubt that code coverage is a good way to show the degree to which the source codes of a program is executed when a specific test suite runs. In general, statement coverage and branch coverage are the most basic form of code coverage. Therefore, the test cases should try to cover all statements and branches of source codes, and try to

find out all infeasible statements and branches. Since code coverage is a white box testing methodology, so it is such important to have a good understanding of each method of each class of a program. For example, it is impossible to achieve full branch coverage without a clear understanding of all situations. Hence, in order to achieve a high coverage, it is necessary to use some ways to increase coverage.

1) The common used method is to cover all branches. This method in not only to increase code coverage, but also to make sure all branches are functional. For instance, in order to help customer to complete check-in, the hotel manager needs to fill in all required information in CheckInPanel interface. If missing one of the required information, the check-in will fail and console will show an error message. The below image display some codes in *checkIn()* method for testing the input is null or not.

```
@Test
public void testCheckIn() {
    /*
     * ID = null, name != null, type != null, company != null,
     * ethernetAddress != null, checkInDate != null, room != null
     */
    Assert.assertEquals("No input can be null", hotelManager.checkIn(null,
            "John", "Standard", "Google", new Date(), false,
            "00:00:00:00:00:00", new Room((int) 6, (int) 1, (int) 2,
                    (short) 1, (double) 1000.0)));

    /*
     * ID != null, name = null, type != null, company != null,
     * ethernetAddress != null, checkInDate != null, room != null
     */
    Assert.assertEquals("No input can be null", hotelManager.checkIn("A12345678",
            null, "Standard", "Google", new Date(), false,
            "00:00:00:00:00:00", new Room((int) 6, (int) 1, (int) 2,
                    (short) 1, (double) 1000.0)));

    /*
     * ID != null, name != null, type = null, company != null,
     * ethernetAddress != null, checkInDate != null, room != null
     */
    Assert.assertEquals("No input can be null", hotelManager.checkIn("A12345678",
            "John", null, "Google", new Date(), false,
            "00:00:00:00:00:00", new Room((int) 6, (int) 1, (int) 2,
                    (short) 1, (double) 1000.0)));
```

Figure 7: Some Test Cases of *CheckIn() in "HotelManagerTest"* class

```
/*
 * ID != null, name != null, type != null, company = null,
 * ethernetAddress != null, checkInDate != null, room != null
 */
Assert.assertEquals("No input can be null", hotelManager.checkIn("A12345678",
        "John", "Standard", null, new Date(), false,
        "00:00:00:00:00:00", new Room((int) 6, (int) 1, (int) 2,
                (short) 1, (double) 1000.0)));

/*
 * ID != null, name != null, type != null, company != null,
 * ethernetAddress != null, checkInDate = null, room != null
 */
Assert.assertEquals("No input can be null", hotelManager.checkIn("A12345678",
        "John", "Standard", "Google", null, false,
        "00:00:00:00:00:00", new Room((int) 6, (int) 1, (int) 2,
                (short) 1, (double) 1000.0)));

/*
 * ID != null, name != null, type != null, company != null,
 * ethernetAddress = null, checkInDate != null, room != null
 */
Assert.assertEquals("No input can be null", hotelManager.checkIn("A12345678",
        "John", "Standard", "Google", new Date(), false,
        null, new Room((int) 6, (int) 1, (int) 2,
                (short) 1, (double) 1000.0)));
```

Figure 8: Some Test Cases of *CheckIn() in "HotelManagerTest"* class

```
/*
 * ID != null, name != null, type != null, company != null,
 * ethernetAddress != null, checkInDate != null, room = null
 */
Assert.assertEquals("No input can be null", hotelManager.checkIn("A12345678",
        "John", "Standard", "Google", new Date(), false,
        "00:00:00:00:00:00", null));


/*
 * ID = "", name != "", company != "",
 */
Assert.assertEquals("ID, name, and company cannot be empty",
        hotelManager.checkIn("", "John", "Standard", "Google",
        new Date(), false, "00:00:00:00:00:00", new Room((int) 6,
                (int) 1, (int) 2, (short) 1, (double) 1000.0)));

/*
 * ID != "", name = "", company != "",
 */
Assert.assertEquals("ID, name, and company cannot be empty",
        hotelManager.checkIn("A12345678", "", "Standard", "Google",
        new Date(), false, "00:00:00:00:00:00", new Room((int) 6,
                (int) 1, (int) 2, (short) 1, (double) 1000.0)));
```

Figure 9: Some Test Cases of *CheckIn() in "HotelManagerTest"* class

2) The exceptions are difficult to cover in testing, because the exception only can be thrown under some specified conditions. Thus, it is a good choice to increase code coverage by testing exception. The image below is an example to show how to test the expected exception in test case. Generally, the execute method *HotelManager()* need to access the database and get data from the database, so if this method cannot find the database in the specified path, this method will throw an exception.

```
@Rule
public ExpectedException exception = ExpectedException.none();

@Test
public void testHotelManagerException() throws Exception {
    new File("5100Hotel.xml").renameTo(new File("Hotel.xml"));
    exception.expect(Exception.class);
    new HotelManager();
}
```

Figure 10: Testing Exception in *"HotelManagerTest"* class

Infeasible statement

1) There are two infeasible statements in the method *getCurrentDate()* which within "*CheckInCommand*" class and "*CheckOutCommand*" class. As the statement which within try {…} will never throw a "ParseExceprion", so this method never need to catch exception.

```
/**
 * Gets the current date.
 * @return the current date
 */
public Date getCurrentDate() {
    try
    {
        return new SimpleDateFormat("dd-MM-yyyy").
                parse(new SimpleDateFormat("dd-MM-yyyy").
                        format(new GregorianCalendar().getTime()));
    }
    catch (ParseException e)
    { // Never happens
        return null;
    }
}
```

Figure 11: method in *"CheckInCommand"* class and *"CheckOutCommand"* class

2) There is an infeasible statement in method *getCommandValues()* which within the *"SearchPanel"* class and *"CheckOutPanel"* class. For example, if variable "room" is equal to null, and then the variable "occupation" will throw an exception called "nullPointerException". Therefore, variable "room" can never be null.

```
searchRoomTable.setModel(new AbstractTableModel() {
                    String[] headers = {"Room No", "Room Type", "Capac
                                        "Occupant Name", "Occupant Type",
                    public int getRowCount() { return searchResults.le
                    public int getColumnCount() { return headers.lengt

                    public Object getValueAt(int r, int c) {
                        Room room = (Room) searchResults[r];
                        Occupation occupation = room.getOccupation();
                        Occupant occupant = occupation.getOccupant();
                        if (room == null) { return null; }
                        switch(c) {
```

Figure 12: method in *"SearchPanel"* class

```
occupiedRoomTable.setModel(new AbstractTableModel() {
                    String[] headers = {"Room No", "Room Type", "Capaci
                                        "Occupant Name", "Occupant Type",
                    public int getRowCount() { return occupiedRooms.le
                    public int getColumnCount() { return headers.lengt
                    public Object getValueAt(int r, int c) {
                        Room room = (Room) occupiedRooms[r];
                        Occupation occupation = room.getOccupation();
                        Occupant occupant = occupation.getOccupant();
                        if (room == null) { return null; }
                        switch(c) {
                            case 0: return room.getFloorNo() + "-" + r
```

Figure 13: method in *"CheckOutPanel"* class

3) There is an infeasible statement in method *UI()* of *"UI"* class. When user click the "X" button on the menu of HMS GUI, the statement "System.exit(0)" will be executed. "System.exit(0)" will terminates the currently running Java Virtual Machine, which means if test case execute statement "System.exit(0)", the JUnit will be terminated. Thus, the statement "System.exit(0)" cannot execute in any test case.

```
    // Add window Listener
    addWindowListener(new WindowAdapter() {
                          public void windowClosing(WindowEvent e) {
                              //store things to database to do
                              System.exit(0);
                          }
                      });
```

Figure 14: method in *"UI"* class

4) There is an infeasible statement in method *addButtonActions()* which within *"ManageRoomPanel"* class. As the method *updateRoomRate()* in *"hotelManager"* class will return "true" if update the room rate successfully, and throw an exception if update failed. So the statement "System.err.print("Error: Room update failed!");" will never be executed.

```
      boolean result = hotelManager.updateRoomRate(command.get
      if(result){
          if(roomInfoPanel != null){
              controlPanel.remove(roomInfoPanel);
              controlPanel.validate();
              controlPanel.repaint();
          }
          getCommandValues();
          editableRoomsTable.setEnabled(true);
      } else{
          JOptionPane.showMessageDialog(null, "room update fai
          System.err.print("Error: Room update failed!");
      }
  } else{
```

Figure 15: method in *"MangageRoomPanel"* class

5) There is an infeasible statement in method *checkIn()* which within *"HotelManager"* class. From the image below, there is unreached statement in red color. This statement 'return "No room is selected"' will execute if condition "room == null" is true. But if room is null, the *checkIn()* method will return "No input can be null". Hence, statement "else if (room == null)" is dead code, and the statement 'return "No room is selected"' is unreachable.

```
public String checkIn(String ID, String name, String type, String company,
                      Date checkInDate, boolean dataServiceRequired, String eth
        if (ID == null || name == null || type == null || company == null ||
            return "No input can be null";
        }
        else if (ID.equals("") || name.equals("") || company.equals("")) {
            return "ID, name, and company cannot be empty";
        }
        else if (room == null) {
            return "No room is selected";
        }
        else if (!type.equals("Standard") && !type.equals("Business")) {
            return "Invalid type";
```

Figure 16: method in *"HotelManager"* class

6) There are two infeasible statements in method *constructMenu()* which within *"UI"* class. Since the exception never be caught when perform action, so the statement "catch (Exception ex) { }" is unreachable.

```
        public void actionPerformed(ActionEvent e) {
            try {
                UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel"
                SwingUtilities.updateComponentTreeUI(UI.this);
                setVisible(false); setVisible(true);
            }
            catch (Exception ex) { }
        }
```

Figure 17: method in *"UII"* class

```
        public void actionPerformed(ActionEvent e) {
            try {
                UIManager.setLookAndFeel("com.sun.java.swing.plaf.mctif.MotifLook
                SwingUtilities.updateComponentTreeUI(UI.this);
                setVisible(false); setVisible(true);
            }
            catch (Exception ex) { }
        }
```

Figure 18: method in *"UII"* class

7) There are eight infeasible statements in method *getOccupantInfoHelper()* which in *"CheckInPanel"* class, and four infeasible statements in method *addSearchInfo()* which within *"SeachPanel"* class. Since the "WindowsLookAndFeel" does not support on Mac OS, so if the test case of "WindowsLookAndFeel" runs on Mac OS,

there is an “UnsupportedLookAndFeelException” will be thrown. Therefore, these four statements are only reachable on Windows OS.

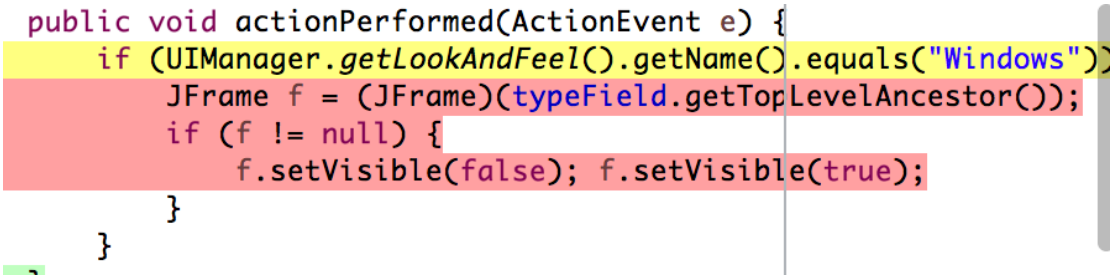


Figure 19: method in *“CheckInPanel”* class and *“SearchPanell”* class

Conclusion

Overall, the test cases for HMS have achieved a very high code coverage. Specifically, achieving 99.5% statement coverage and 98.3% branch coverage on Windows OS (98.5% statement coverage and 94.6% branch coverage on Mac OS) have been higher than expected.